ESD-TDR-65-65

AD 614634

~~COPY 2 OF 3~~
~~HARD COPY $2.00~~
~~MICROFICHE $0.50~~

Technical Note 1965-13

J. A. Feldman

Aspects of Associative Processing

21 April 1965

Prepared under Electronic Systems Division Contract AF 19(628)-500 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts

20040826029

[redacted]

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

ASPECTS OF ASSOCIATIVE PROCESSING

J. A. FELDMAN

Group 23

TECHNICAL NOTE 1965-13

21 APRIL 1965

LEXINGTON MASSACHUSETTS

ASPECTS OF ASSOCIATIVE PROCESSING

ABSTRACT

Several aspects of information processing on a computer with an associative memory are discussed. The first section describes a source language for use with such an associative processor. An efficient way of simulating an associative processor on conventional computers is the topic of the second section. The conclusion contains a discussion of the underlying assumptions as well as several suggested applications. The applications to artificial intelligence and natural language processing indicate the greatest promise.

Accepted for the Air Force
Stanley J. Wisniewski
Lt Colonel, USAF
Chief, Lincoln Laboratory Office

TABLE OF CONTENTS

I. A PROGRAMMING LANGUAGE FOR ASSOCIATIVE PROCESSING

An associative processor is any computer or computer program which performs the functions of the mythical associative memory. After a brief discussion of the essential features of the associative memory, we will consider several aspects of non-numeric information processing with such a device.

The first topic will be an intermediate level, programming language designed for use with an associative processor. In the second section we describe a program for the TX-2 at Lincoln Laboratory which is an efficient associative processor. The final section contains a discussion of possible applications of associative processing as well as a critical discussion of the first two sections. It is here that we will distinguish arbitrary design decisions from those we believe to be significant.

The distinguishing feature of an associative memory is that is has no explicit addresses. Any reference to information stored in an associative memory is made by specifying the contents of a part of a cell. All cells in the memory which meet the specification are referred to by the statement. A standard coordinate-addressed memory can be thought of as a special case in the following way: Each cell of the associative memory will be a pair consisting of a conventional cell and its address:

associative cell

address	conventional cell

To access a cell in this special associative memory, one specifies the contents of one particular part (the address field) of a cell. In the general associative memory a cell can be accessed by specifying the contents of <u>any</u> part of the cell.

The following example may help point out the importance of this seemingly minor difference. Suppose one were to store the contents of a telephone directory in a computer memory. There are several ways in which one could organize the data so that the telephone number of a given person could be found fairly quickly. However, the problem of going from a telephone number to its owner is rather difficult. One could, of course, enter a second directory ordered by numbers rather than names, but this entails representing the same information twice in the memory. There are several other possibilities; all of these lead to compromises between inefficiency in time and inefficiency in storage. In an associative memory either question could be answered in one memory access and without any redundant information being stored in the memory.

This example is so clear and striking that it should be suspect. First of all, it is often the case that one person has several telephone numbers or that several people are listed for the same number. This is the so-called multiple-hit situation and is at the root of most of the problems encountered in using associative memories. We will return to this question repeatedly in the sequel.

To illustrate the consequences of this and other difficulties we will need a more comprehensive example. For this purpose suppose the telephone book is expanded to an Orwellian directory containing the name, address, telephone number, social security number, and family ties of each entry. The problems inherent in representing this directory are qualitatively different from those in the simple example.

When there were only two objects, a name and number, the representati in the associative memory was straightforward. The obvious extension of this technique to the larger problem would lead to a memory with words wide enough to contain the five pieces of information. One difficulty with this scheme is that it is very wasteful. For example, a grocery store can be expected to have an address and telephone number, but no social security number or family ties. By using a different field for each attribute we guarantee that a large part of the memory will always be wasted.

A more serious problem arises in attempting to represent family ties. There is no reasonable fixed word size which will be large enough to represent the family relationships of the person with the most ties. Further, the representation of a family tie requires two pieces of information: a person's name and his relation to the original person. All of these difficulties are inherent in the problem and characterize the type of information structure with which this paper is concerned.

Thus, we are dealing with problems where there are a large number of <u>objects</u> of different kinds (people, businesses). Each of these objects will have certain <u>attributes</u> (address, family ties). We assume that there are no general rules for deciding which attributes apply to a particular kind of object. For example, the attribute "SOCIAL SECURITY NUMBER" applies to most, but not all, people. Each attribute which applies to a given object will have a set of <u>values</u>. To represent such an association we will use the notation:

1) Attribute(Object) = Value

The notation above can be motivated by considering the attribute a

a function which has objects as its arguments and produces values. In this scheme a representation of the large directory might contain triples like:

ADDRESS(John Doe)	=	201 Main
PHONE(John Doe)	=	862-3926
SON(John Doe)	=	Don Doe
WIFE(John Doe)	=	Donna Doe

Notice that we have used the attributes SON and WIFE rather than the original form FAMILY which would have led to:

FAMILY(John Doe)	=	Don Doe, son
FAMILY(John Doe)	=	Donna Doe, wife

This rearranging of information to correspond to the triples of the basic form (1) will reoccur frequently below.

The remainder of this section will be devoted to a discussion of a programming language for use with an associative memory. The presentation itself will not depend on the implementation chosen for this associative memory. The language is currently running on an associative memory simulated on the TX-2 at Lincoln Laboratory.

The essential feature of any associative processor is the facility for implicitly specifying operands. One can use constructs like SON(John Doe directly in statements. This ability becomes very powerful when nesting to arbitrary depth is also permitted. We will, for example, allow operands which express:

"the telephone number of the son of the man whose brother lives with Mary's Aunt".

This construction is already fairly difficult to express in English and there are some constructs which are practically inexpressible. The essential difficulty is that natural language does not have sufficient ability to name intermediate results. This is accomplished in formal mathematical systems through the use of variables. We will make extensive use of the properties of free and bound variables to help express complex associations.

All operand specifications will be based on the form (1) which we abbreviate:

2) $A(O) = V$

The basic operands are formed by replacing one or two of the symbols in (2) by variables, usually W, X, Y, or Z. Although the variants of the form (2) may entail complex processes, they are syntatically operands for greater programming ease. For example, SON(John Doe) = X represents the set of all sons of John Doe. This is the most common case but is only one of the six possibilities.

F1) $A(O) = X$

F2) $A(X) = V$

F3) $X(O) = V$ (cont)

F4)	$A(X) = Z$
F5)	$X(Z) = V$
F6)	$X(O) = Z$

There is, in addition, the case FO where all the positions are specified. This operand form is just a test for the occurrence of the fully-specified association in the memory. These seven forms correspond to the basic searches available in a three-field associative memory; the symbol F may be thought of as mnemonic for FIND. In terms of the relationship between John and Don Doe these operands are:

NAME	ASSOCIATION	MEANING
F1	SON(John Doe) = X	Sons of John Doe
F2	SON(X) = Don Doe	Father of Don Doe
F3	X(John Doe) = Don Doe	Relation of John to Don Doe
F4	SON(X) = Z	All father-son pairs
F5	X(Z) = Don Doe	All Associations with Don as <u>Value</u>
F6	X(John Doe) = Z	All Associations with John as <u>Object</u>
FO	SON(John Doe) = Don Doe	The association itself, if true

The programming language described here will be based rather strongly on the properties of the operands FO......F6, which will be treated as primitives. These are actually implemented by generators [5] in the simulated system described in the next section. The language presented here, however, will not depend on any particular implementation of the primitives. This is somewhat of a departure from the previous work in this area.

The existing literature on programming an associative processor is mainly concerned with special applications and is at the machine language level. Without questioning the value of this work, one can assert that the wide-spread acceptance of associative memories will depend on their use in general-purpose machines and with appropriate problem-oriented languages. Further, in developing such high-order languages, we may provide valuable information to the designers of hardware associative processors.

It will be important to keep in mind that the associative language (AL) described here is meant to be compiled. There will, of necessity, be large run-time subroutines, but the advantage over an interpreter-oriented language still seems significant. The language will make no explicit use of the considerable parallel processing ability available in some of the proposed associative memories. Finally, the language is strongly oriented towards applications in artifical intelligence and may not be well suited to other areas.

Besides the operands F0......F6, there will be three basic operations in the associative language. These operations, SET, TYPE, and ERASE, correspond to the primitive functions WRITE, READ, and DELETE of a non-destructive read-out memory. To illustrate their use we will refer to the example in Appendix A. Listed there is an actual run of the TX-2 associative processor dealing with the information in a simple picture. A copy of Appendix A is at the back of the paper and may be detached.

In the example of Appendix A a number of special conventions are followed. All attribute names end with a period ".". The only variables used will be W, X, Y, and Z for objects and values and W·, X·, Y·, and Z· for attributes. The lines beginning with circled numerals are comments and all of the program's responses are indented.

The first four lines of Appendix A contain commands putting some initial information into the memory. The operation SET requires a form with all positions fixed (FO). Upon execution the SET command places the specified association in memory if it is not already there. Thus, the first line of the example records that CIRC and S01 are both to the left of TRI. The statement RUN causes the compiled code to be executed and the translator to be reinitialized. Appendix B contains a Backus Normal Form approximation to the syntax of the associative language, AL.

Notice that only some of the positional relationships among objects in the picture have been placed in memory. In the course of the run various AL statements are used to expand the information. For now, however, our goals are more modest.

The statement ① of Appendix A is an encoding of the question "How are SQ2 and CIRC related?". The AL statement

```
TYPE W·(SQ2) = CIRC, W·(CIRC) = SQ2, RUN
```

carries out this operation. The function TYPE can accept as its operand any of the forms FO......F6; it causes all the specified associations to be typed. The syntax of ERASE is exactly like that of TYPE. An example of its use can be found on line ⑦ where the LINE is completely removed from the picture.

The basic operations may be combined in various ways using sequencing statements. The most elementary type is the simple FOR statement:

```
FOR  < form >     < statement sequence >     END
```

The AL statement ② is an instance of a simple FOR statement. We will consider it in some detail because it points out the use of bound variables. The statement is:

FOR ABOVE · (CIRC) = X TYPE LEFT · (X) = CIRC END, RUN

The form in this statement is of type F1 and specifies all objects above the circle. Since the variable X occurs within the scope of FOR, it is bound until the matching END. Thus, the operand of TYPE has all positions fixed (F0) and is a test for the occurrence of the fixed association in memory.

The control flow of this statement models closely the generators of IPL-V [5]. All figures which are above the circle are found. Each figure which is so generated is tested to see if it is also to the right of the circle. If so it is printed and, in either case control returns to the generator for another figure to be tested. This correspondence between bound variables and generators was independently utilized in the DEACON effort and is discussed in [11].

A more complicated FOR statement can be found in ③ of Appendix A. This statement expands the memory to explicitly include results of the transitivity of ABOVE. By replacing the SET in ③ with ERASE one could construct a statement which would contract the memory on transitivity. Statements such as these allow an AL user to dynamically trade space for time as problem needs dictate.

Another expansion of the memory is accomplished by the statements in example ④ . This example is motivated by the fact that the LINE inherits the positional properties of SQ2 which it is inside. The AL representation uses two statements, one for associations with SQ2 as object and one with it as value. A complication arises in the second of these statements which necessitates the introduction of another kind of conditional clause.

In the second statement of ④ we are applying all associations with SQ2 as object to LINE. The difficulty is that one of these associations is:

$$\text{INSID} \cdot (\text{SQ2}) = \text{LINE}$$

and if the obvious course was followed, the statement ④ would yield

$$\text{INSID} \cdot (\text{LINE}) = \text{LINE}.$$

Under the assumption that INSID means properly inside, this is incorrect. To avoid the undesirable result we have inserted the extra qualifier:

$$\text{IF} \quad Y \quad \Delta \quad Z$$

where Δ means "differs from". The nature of the IF clause is essentially different from that of the other constructs described above.

The difference is that the expression "Y Δ Z" deals with objects outside the context of the associative memory. Since a very limited number of operations are available within the memory, there must be provisions for dealing with associative entities in conventional ways. One important test of any associative language is the ease with which it allows mixed statements to be expressed.

In the current version of AL we have implemented a number of arithmetic and relational constructs to indicate how this might be done. To help illustrate the use of these constructs, we have extended the example in Appendix A to include information about the areas of the figures.

The circle, triangle and squares are all given named areas; the area of the line is set to the constant 0. Then the named areas are assigned values by statements like:

$$1 \longrightarrow \text{ATRI.}$$

This causes ATRI to be marked as having a numeric value as well as assigning that value. The statement in example ⑤ illustrates the use of the numeric values of associative operands. There is also in ⑤ a use of the operator TYPOB which acts on an object, as opposed to TYPE which operates on an entire association. An execution of TYPOB will type out the name of an object and if it has a value, its value.

In example ⑥ there is a combination of an associative and a numerical test. As described in Appendix B these two types of test can be nested arbitrarily in AL. There are also some primitive arithmetic expressions in the language, but none appear in the examples.

The associative language, as described above, is not yet rich enough to do much useful processing. There are two kinds of extensions required: involved and trivial. The involved extensions require detailed consideration and are best left until after the discussion of implementation in the next section. The trivial extensions are conceptually trivial but require non-trivial amounts of work; these were not implemented because there were neither people to do them nor user pressure to get them done. They will be mentioned briefly here.

One of the most obvious extensions would be full arithmetic capability. The language should certainly have control features such as labels and transfer commands. The related concept of subroutines is also needed, but falls among the involved examples to be discussed later.

The restriction to fixed name types for attributes and variables could be overcome by incorporating ALGOL-like declarations. It would also be possible to make the language easier to use by taking advantage of intuitive preferences for the Form F1. For example, one could allow

$$\text{AREA} \cdot (\text{ABOVE} \cdot (\text{TRI})) > \text{AREA} \cdot (\text{TRI})$$

as a legal expression. We have not found a natural way to extend this abbreviation to the other F forms.

Most of the extensions described here could be carried out by imbedding AL in an existing programming language. After discussing our implementation of AL, we will consider the imbedding problem in some detail.

II. A SIMULATED ASSOCIATIVE MEMORY

The associative addressing simulator for the TX-2 is a descendent of the well-known scrambling or hash-coding schemes. The basic notion is that objects be given internal names which correspond in some simple way to their location in memory. Consider this example frequently found in compilers for algebraic languages. An identifier in, say, ALGOL 60 is any string of letters and digits starting with a letter. For the internal purposes of the compiler it is required only that the identifiers be mutually distinguishable. By assigning a unique low integer to each identifier, the compiler writer derives several advantages.

If the reserved words of the language are chosen to be the first n integers, a simple test for reserved words is available to the syntax phase of the translator. More significant, from our point of view, is the elimination of some table-lookup routines. One simply puts the external names of variables in a sequential table starting at EXTNAME. Then when printing a variable whose internal name is N the compiler accesses the contents of EXTNAME + N. One could also use another table at PROPERTY for various properties of the identifier.

Let us temporarily abandon this example and reconsider the nature of an associative memory. For the present it is convenient to assume that the data is arranged in three fixed fields in a word. The more general case of associating on any set of bits will be considered later. The primary feature of the memory is that all of its cells may be tested in parallel for the occurrence of a particular pattern in one or more of the fields. This is the accessing of data by its <u>contents</u> rather than by a particular address and has given rise to the alternative name content-addressed memory (CAM) for the associative memory.

The three fields in a word will be called the attribute, object and value fields and the triplet itself called an association.

3) association

attribute	object	value

This notation is identical with that of the first section and is also common to several list-processing languages. This similarity of notation is not accidental; the description lists of IPL were the first systematic attempts to simulate an associative memory. The great usefulness of the description (property) list concept as well as its limitations will be discussed below.

Let up now reconsider the example of the compiler symbol tables. Let each internal name be considered an object. Then for the attributes EXTNAME and PROPERTY we have a convenient associative scheme for attaining their value at any object. This use of memory corresponds to considering a cell of conventional memory as an extended word of the form

4) cell

address	value

Under the further assumption that EXTNAME and PROPERTY have enough low-order bits equal to zero we can rewrite the address as "attribute, object" obtaining

5) cell

attribute	object	value

in obvious parallel to (3).

It is useful to examine the restrictions inherent to this example. There are only two attributes and each is applicable to every object. Any attribute applied to any object has one and only one value In addition, the scheme is only efficient for asking the question ATTRIBUTE (OBJECT) = X. For example, there is no direct way of obtaining the internal name of an identifier from its external name. Many of these restrictions have been removed in existing list-processing languages. The scheme used in IPL-V is typical.

In IPL-V the basic entity which corresponds to an object is the list. Each list may have an attached description list which in turn is a sequence of attribute-value pairs. To find the EXTNAME of an identifier I, one would access the description list of the list named I. Then a sequential search would be made for the attribute EXTNAME and the next cell on the list taken to be the value. The value may itself be the name of a list and thus multiple values are permitted. This scheme is more general than that in the compiler example, but still has serious drawbacks.

There is, in the description list structure, no way of directly answering inverse questions except by including the inverse of each association explicitly. In addition, since the attributes must be searched sequentially, the time involved for large structures becomes prohibitive. Finally, there is no provision at all for ascertaining the attribute linking a particular object-value pair. Despite these weaknesses, description lists are among the most useful features of the list processing languages. In fact, many of the really large programs in these languages rely heavily on esoteric uses of description lists. This accomplishment in the face of adversity has suggested a search for better associative memory simulators.

In the compiler case we are able to get values quickly, but the number of attributes was limited. In IPL-V there was an unlimited number of attributes available, but answering the question ATTRIBUTE (OBJECT) = X could involve considerable sequential search. The attempt to overcome this dilemma has given rise to the various scrambling and hash-coding schemes. The idea is to apply a transformation to the internal names of ATTRIBUTE and OBJECT which will yield the address of the appropriate value. Since this is unattainable in practice, attempts were made to reduce the expected value of the number of accesses to arrive at the value of a given association.

Although there has been much discussion of such techniques, very little has been published in this area. We will describe briefly an unpublishe paper by Newell [6] which is the most extensive treatment of the problem we have seen.

In Newell's scheme associations will each occupy a cell of conventional memory. Each attribute, object and value will be assigned a random number of the same size as an address. To find a cell for A(O) = V, one adds A and O. Since it may be the case that A + O equals some A' + O', each cell must have a linked list of overflow cells. It is also assumed that associatior are all single valued.

Under these assumptions it is easy to compute the expected number of overflow cells as a function of the ratio of memory filled. Newell performs this analysis and uses the results to compute the expected number of accesses for the basic associative memory operations. His conclusions are that the simulation loses a factor of two in speed and four-thirds in storage over a hardware associative memory of the same basic speed.

The scheme presented here is an extension in several directions of Newell's simulator. In his scheme only the question A(O) = X is directly answerable and, further, it is constrained to have only one answer. The attempt to generalize on these restrictions has led to the development of a rather different simulator.

The particular memory structure chosen here was determined by a number of initial decisions, some of which are open to question. Among the most critical was the choice of different logical types for attributes than that chosen for objects and values.

At the beginning of the project a decision was made to have the attributes be of a higher logical type (functions) than the objects and values. Although this would not be true in a hardware associative memory, it has proved to be useful. There have also been some difficulties which will be discussed in section III. When there is no danger of confusion we will refer to both objects and values by the generic name, object. Thus, an association may be thought of as specifying a function from one object to another.

A further point to notice is that there will always be fewer objects than there are cells of the associative store. This is because each object must be involved in at least one association in order to be in the store at all. Since, in addition, each associative cell will require more than one memory word, the number of bits used in an object name can be less than that in an address. Further, since attributes and objects appear in different contexts, we can use the same name for one of each without confusion.

One can also argue for dispensing with random numbers as internal names. This is justified if the associations are themselves fairly random with respect to the pairing of attributes and objects. As we will see it would require some odd periodicities to violate this assumption for our sche

By assigning internal names sequentially we also gain some peripheral advant

which will be discussed below.

Under these assumptions one can build a simple associative simula

The TX-2 has a 36-bit word and we currently use the first 32,000 registers as the associative store. Using two words per association, this allows roughly 16,000 associations. For general use it seems that a maximum of 4,000 objects (12 bits) is reasonable. Since there are presumably fewer attributes, we use only 500 (9 bits) as the upper bound on attribute names. Thus, the entire association can be packed into one machine word:

9 bits	3	12	12
attribute	tag	object	value

The other word of the associative cell will be used for the inevitable links. As everyone knows, tight packing leads to slow processing and our system does suffer somewhat in speed from this effect. However, we felt that maximum utilization of core storage is one of the most important requisites of an associative scheme.

The next step is the choosing of the transform used to compute the address of an association. The only criteria that seem important here are that the transform be simple and that it not be biased. A biased operat is one, like logical AND, that is more likely to produce zeros (or ones) as a result, given random operands. We chose the unbiased operator EXCLUSIVE O ($\oplus$, partial add) because it is its own inverse and this property might prove to be useful someday. Finally, it will be necessary to shift one of the operands so that the result is large enough to address memory. The shift is chosen so that the result has enough bits to address the largest memory anticipated and a mask is then used to address a smaller associative store when desired.

In our scheme an attribute (9 bits) and an object (12 bits) are transformed into an 18-bit number by shifting the attribute left nine places and partial-adding the object name. The TX-2 has 9-bit byte commands so this all comes off fairly smoothly. At present this 18-bit number is masked to eliminate the three high-order bits and the lowest bit. The result is the address of the even register of one of the association cells. For example, if attribute 222 of object 6543 has as its value object 1234, the following takes place: The numbers are all in octal form.

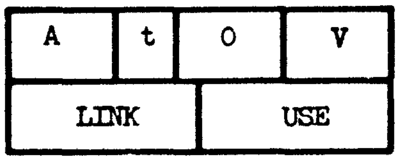
First 222000 $\oplus$ 6543 = 224543 is computed. This is masked to yield the address 24542, which we assume is unoccupied. The top of cell 24542 is filled in as follows:

24542	222	0	6543	1234

If the association cell is already full, various complications arise. There are three different ways in which cell 24542 could have been filled prior to this operation. First of all, if 222 of 6543 has another value, this would automatically occupy the same cell. This situation, called <u>multiplicity</u>, is the multiple-hit situation and is a problem in every associative scheme, hardware or software. It is also possible that another association, such as 220 or 4542, would yield the same cell 24542 under the transform. This is called <u>overlap</u> and its statistics are a measure of the performance of any simulated system. Finally, the cell 24542 may have been used as a free-storage cell to handle the overlap or multiplicity of another cell. This situation, <u>conflict</u>, will require moving the contents of 24542 to a new cell from free storage and is inordinately costly.

Because of the overlap and multiplicity problems, there must be a provision for linking extra cells to the one produced by the transform. This requires a free-storage list and a LINK position in every cell. The form of an associative cell will then be:

association



where A, t, O, V stand for attribute, tag, object, and value, respectively. The half-word marked USE holds a link in the linked list of all uses of an object in the V position of a cell. It is this feature which makes it feasible to answer inverse questions. The heads of the USE lists for the individual objects are kept in a sequential table starting at VUSE.

The most important subroutines in the simulator are those which implement the operands FO......F6. These arise from the replacement of zero, one or two of the symbols in

$$A \cdot (O) = V$$

by variables. The resulting forms correspond to the seven possible operand specifications in a three-field associative memory. The only other routines needed to simulate a pure associative memory (no processing ability) are SET, TYPE, and ERASE. Thus, to simulate an associative memory one need write only a small number of fairly simple programs.

The work reported has proceeded along slightly different lines. Our main interest in the simulated associative memory was as a vehicle for the language described in the first section. For this reason the routines used in the simulator are somewhat different than they would be for a simulated pure memory. Therefore, it will only be possible to make rough estimates of the efficiency of the system as a memory. The rest of this section presents a discussion of these routines, some performance figures, and some comments on the efficacy of simulating associative memories.

The six operand specifications F1......F6 are each implemented by two subroutines, one recursive and the other not. There is a degredation in performance among them which corresponds to our estimates of their frequency of use.

The routine F1 is used to retrieve operands of the form $A \cdot (0) = X$, the most frequent form. The internal names of A and 0 are transformed into an address as described above. If the addressed cell is not empty and not used as a spare, its contents are tested for the occurrence of A in the attribute portion. If it matches the cell is an answer and is output. Multiple values will be in sequence on the linked list through the LINK section of the cell. If the attribute portion of the cell is not equal to A, overlap has occurred in that cell. In this case the list through LINK is searched for a cell with an attribute equal to A (Positive answer) or greater than A (Negative answer). A flowchart of this routine is given in Chart 1 on page 22. Although this looks like a lot of processing, it actually is quite fast as Table I will show. We will consider this flowchart in some

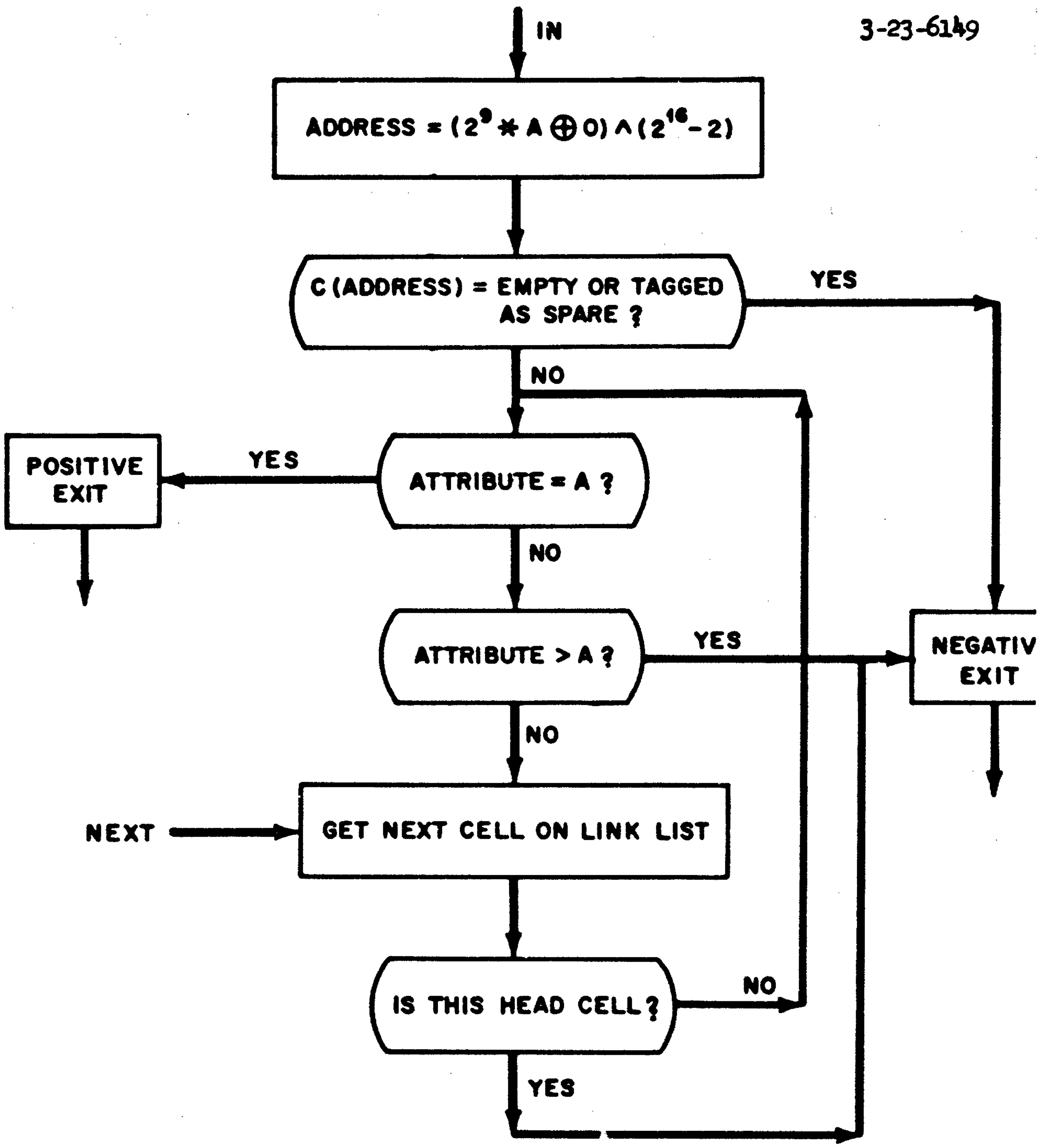

3-23-6149
IN
ADDRESS = $(2^9 * A \oplus 0) \wedge (2^{16} - 2)$
C (ADDRESS) = EMPTY OR TAGGED AS SPARE ?
YES
NO
POSITIVE EXIT
YES
ATTRIBUTE = A ?
NO
ATTRIBUTE > A ?
YES
NEGATIV
EXIT
NO
NEXT
GET NEXT CELL ON LINK LIST
IS THIS HEAD CELL ?
NO
YES

CHART 1

FLOWCHART OF THE ROUTINE F1

detail since much of the discussion will apply to all the F routines:

The routine F1 has two entrances and two exits. The initial call is to the top of the flowchart and leaves a return address or mark. The negative exit is to mark, and the positive exit to mark + 1. The second entrance (Next in the flowchart) is used for successive values if these are desired and present. The recursive version differs in that it saves several things before leaving through the positive exit. All of these details are common to the routines F1......F6 and none of them concern users of the language described in the last section.

The routine F2 is used to generate answers for A · (X) = V. The USE list of V is searched sequentially for the appearance of A in the attribute section of a cell. The USE links are also in order of increasing attribute numbers, so the loop in F2 is identical to that shown in the flowchart for F1, except that the USE list is followed.

The other routine for a one variable search is F3; X · (O) = V. This is implemented by searching the USE list of V for occurrences of O in the object field. The occurrences of O are not ordered so this routine is slower than F1 or F2.

The routines F4, F5, and F6 answer questions where two variables are combined with one known entity. The routine F5 for X · (Y) = V is particularly simple. Since each object has a list of all its uses as a Value, F5 merely follows this list.

The most costly (and least likely) questions is F4, A · (X) = Y. This routine is a loop, applying F1 to every object in turn. There are several ways of alleviating this situation if F4 turns out to be more frequent than expected. The routine F6, X · (0) = Y is also a loop. This one applies F1 in turn to every attribute in the system. Presumably, there are far fewer attributes than objects so this is not as slow as F4.

The routine F0 is different in that it can have at most one result. This case, A · (0) = V, has everything fixed and simply leaves by the positive exit if the specified association is in the store. The seven F routines implement the READ function of an associative memory.

The routine to write an association (SET) operates on the form F0, where all parameters are fixed. It uses the routine F1 to find the proper place in the memory for the association and to ascertain that it is not already present. The routine F1 also provides the information needed to set the LINK portion of the cell. The USE tie is placed appropriately by using F2 to find the predecessor of the new association on the list at VUSE + V.

One difficulty in writing is that the cell found by F1 may have been used as a spare. This is the conflict case and should be minimized. For this reason the free storage list starts at the top of the memory, while the associations build up from the bottom (because names are chosen sequential This insures good performance of a mostly empty memory while allowing the use of every cell when the memory is full. Actually, there is now a separate area of 12,000 locations directly above the associative store which is used first as free storage.

The ERASE routine can be used in connection with any of the generators F0......F6 in the recursive version. Both the LINK and USE lists are circular (the last element points to the head element) so that the predec of the cell to be erased are available. The head of each list can be determi from the top of the association cell, so only half the list must be searched on the average. It would be more efficient to have seven different ERASE routines, but this has not been done.

In Table 1 the minimum access times for the various routines described above are listed. The times are minimal in the sense that each routine could involve list searching in the worst case. The length of the lists differs from case to case. For F1 the length will be the number of overlap cells with attr. < A. For F0 the length will be that of F1 plus the multiplicity of A. The length needed for F2 equals the number of uses of V with attr. < A. The routine F3 has no ordering available and must search the entire USE list of V regardless.

The routines F4 and F6 are loops using F1 repeatedly. The length of the loop is, for F4, the number of objects used and, for F6, the number of attributes used. The times for SET and ERASE are not listed because there was no attempt to optimize these routines. They are currently taking 200 μ seconds each on the average. The TX-2 has a memory rate of about 5 μ seconds.

MINIMUM ACCESS TIMES (μ seconds)

	NON-RECURSIVE		RECURSIVE	
	FIRST	NEXT	FIRST	NEXT
F1	50	40	90	120
F2	40	40	80	120
F3	50	50	90	130
F4	*	*	*	*
F5	50	20	90	80
F6	**	**	**	**
F0	50	—	—	—

* F4 uses F1 on each object in memory
** F6 uses F1 on each attribute in memory

There are several interesting features in this table. The most noticeable is the high cost of recursion. The recursive version of a routine is needed when another F routine will be called before the first terminates. This can be determined by the compiler and is one justification of compiling the associative language. A hardware associative memory must also have a way of handling multiple hits and this would probably be even more costly relative to access time.

The other point is that the times are sufficiently small to raise questions about competing with hardware techniques. For a computer with list following and stacking instructions these times would be greatly reduced. We are currently studying possible modifications to the TX-2 which would speed up the entire system appreciably. It should be feasible to build a system which loses a factor of about two in storage and three-to-five in time against an associative memory of the same basic speed.

Assuming this is possible, it is debatable whether it will pay to build hardware associative memories for general-purpose use. The most obvious advantage of the simulated scheme is cost, but there are a number of others. Given the current state-of-the-art, another significant advantage is size. The most ambitious hardware systems are designed for at most 16K memories. With the new 1,000,000-word conventional memories, one could construct associative systems with hundreds of thousands of cells.

Another advantage of simulated memories is flexibility. For example, it would be trivial to add threshold searches of various sorts to our scheme. A related feature is that one could combine some memory operations with other processing and thus reduce the time disadvantage. One of the purposes of the language of section I is to help discover which combinations are the most useful.

In addition, there are some complications in the use of hardware associative memories which are not immediately obvious. Most of these are concerned with the multiple-hit problem. For one thing it is quite difficult in most systems to even generate the multiple answers in order. Further, if much processing must be done outside the associative memory the saving in the time for the first access may be a negligible part of the total. Even in systems with considerable parallel processing ability the result is often in the wrong field and must be moved. Finally, there is no direct provision for processes involving recursion and tree-structured data. Since almost all artificial intelligence work depends on these abilities, certain additions to the basic scheme would be necessary.

A further point concerns the limitation of the simulated system to three fixed fields. While it seems that a hardware memory would permit searching on any bit pattern, this is not true in practice. The difficulty is that the entire memory is searched in parallel and overlapping fields would lead to false positive responses to some questions. This is just another instance of the scheduling problem and restricts the hardware associative memory to fixed field comparisons. One does have the ability to choose different fixed fields for various problems, but this is not nearly as powerful as searching on any bit pattern.

Some corroboration for our point of view may be found in Fuller [2]. In this paper he considers the applicability of an associative processor with considerable parallel processing ability. Although he compares tomorrow's associative processor with yesterday's conventional machine, he concludes that for artificial intelligence list processing seems to be best overall.

Thus, there is some evidence that large associative memories may be competitively simulated on conventional computers. There are still many applications (Fuller [2]) where hardware associative memories are unquestionably superior. One use of current interest is in page-turning memories, especially in time-shared systems. While research in large , general-purpose associative memories should also be encouraged, there seems to be no reason to wait for it.

III. APPLICATIONS AND EXTENSIONS

In this section we will describe some possible uses of associative processors and also discuss critically the results of the first two sections. There are two distinct reasons why applications could arise from these results; the efficiency of the processor and the convenience of the associative language. The topics of this section will be treated in the following order: design of the simulator, applications of both kinds, improvement to the language.

The simulated associative memory is designed to replace some of the operations in existing list-processing languages. We will attempt to resolve some of the known problems of associative schemes as they apply to our model. One of the principal objections to hash-coding schemes has been the large memory commitment. Even for small problems, one had to allocate the entire associative store. We have attacked this problem at three levels. Since internal names are assigned sequentially, the memory used grows fairly regularly with the size of the problem. Further improvements can be attained by proper choice of the mask.

If the size of a problem is known in advance a mask of appropriate size can be chosen for the transform on page 19. Assuming random interconnections, one has the ability to use less storage at the cost of more overlap in a statistically predictable way. Finally, the system can be set to automatically expand the size of the associative store (bits in the mask) as the problem grows. This involves moving some association cells, but the information necessary for this is contained in the cells themselves.

Another objection to the associative schemes has been the lack of an adequate language. This problem, although not solved completely here, has been shown to be within reach. Aside from these basic criticisms, there are some alternative implementations to be considered.

The first set of alternatives concern purely practical questions. By linking multiplicity and overlap cells on the same list, we incur longer search times. The alternative would be to add a level of indirectness for all multiple values, leaving only overlap cells on the LINK chain. The relative cost of longer searches vs. an added level of indirectness is problem dependent and we don't know enough about the mix to force a conclusion.

Another question of efficiency arises in connection with the LINK and USE lists. They are now circular so that ERASE involves two searches for predecessors. By using two-way links (ala SLIP) one could eliminate these searches. We chose not to do so from a belief in the rarity of ERASE statements and a feeling that storage was the critical parameter. Newell's decision to use circular lists was based on the assumption of short lists and does not apply here.

Some more serious design decisions were involved in the treatment of inverses. Newell chose not to include any explicit inverse operations both in IPL-V and in his associative scheme. We feel that the operand flexibility of the associative language is one of its most important features Being unable to discuss the problem objectively, we leave it for the reader's consideration.

As we mentioned in section I, the logical type of attributes is a key issue. The decision to distinguish attributes from objects was initially made to avoid paradoxes like those which plagued mathematical logic. The resulting practical advantages have included the double use of internal names, use of fewer bits for attributes, and the ability to do some error checking.

The disadvantages arising from this decision did not appear until we attempted to treat mixed expressions. The problem here is that each non-associative routine would need two forms, one for objects and the other for attributes. For example, a better version of $\textcircled{4}$ in Appendix A would have "IF Y Δ Z" replaced by "IF W $\cdot$ Δ INSII$\cdot$" but the latter was not available.

This issue is further clouded by the fact that it would be convenient to allow declarations of the properties (transitivity, symmetry) of attributes which will be effective at compile time. Finally, it is always possible to choose an identity element θ such that for each attribute, A, there is a unique object, a, with A $(\theta) = a$. Thus, there is a natural correspondence available between attributes and objects. This suggests that perhaps the best solution is to have the first 500 (in our case) objects also be attributes with context determining which use is intended.

We will close our discussion of the simulator with two questions of a statistical nature. The first concerns the assumption of independence between objects and attributes. We recall that the address of A $\cdot$ (O) = V was computed by:

$$\text{address} = ((2^9 * A \oplus O) \wedge 2^{16} - 2).$$

Then, assuming even numbers for all object names, the relation A(O) = A' (O') holds if and only if:

O = O' modulo 2^9 <u>AND</u>

A = A' in the middle three bits <u>AND</u>

(cont)

the lower 3 bits of A partial added to the highest

3 bits of O equals the same function of A' and O'.

We assert that for normal problems this is a random occurrence among even numbers $< 2^{15}$ and thus has probability $1/2^{14}$. Including odd-numbered objects, doubles the probability of overlap so in our system all even numbers < 4000 are used first for object names.

If the overlap probability is indeed the same as for random addresses, we can consider using Newell's statistics to analyze our system. However, because of the multiple values it does not directly apply. In fact, the performance of the system depends on the expected number of multiple values and is problem-dependent. With these reservations we note that the performance of Newell's system did not depend appreciably on the percentage of memory occupied.

While it is clear that many open questions remain, the associative simulator is sufficiently well developed to be used in many current problems. The main advantage of the associative processor is that for certain types of data it takes less space and much less time than any known alternative scheme. The most suitable problems are those where the data includes large numbers of unpredictable relations between objects of significantly different kinds. For neater data structures one of the list processing languages or a block-structure language such as CORAL [7] may be a better choice.

One problem area where the data is evidently not neatly structured is natural-language processing. Within this area, the most obvious applicatic is in the so-called question-answering programs. Since an extensive review of this work has been published recently by Simmons [10], we will treat it only superficially. The basic problem in question-answering is the retrieval of ·esponses to questions expressed in natural language. It is usually assumed that the data base is also given originally in natural language. This work has also been referred to as <u>fact retrieval</u> as opposed to information retrieval, which has come to mean the recovery of pertinent documents.

Most of the recent work in question-answering has been done with the description list processes of one of the list languages. Even in their present rudimentary form, the results of this paper should provide a significant improvement in the capabilities of these systems. The examples in Appendix A are typical of some of the internal processing that might be done in question-answering programs. The greatest difficulties in question-answering do not, however, occur in the internal processing. The most difficult problem, extracting the meaning of English sentence, also appears in many other problems.

The entire question of the semantics of natural languages has received much attention recently. Small, computer-oriented, semantic models have been used in the work of Jane Robinson [8] and Thompson, et al [11] among others. A more extensive, if not immediately applicable, treatment can be found in Katz and Postal [4]. There are, by now, several testable theories of natural language semantics. Associative processing seems to apply in various ways to these systems.

A basic property of the various semantic theories is close attention to the various attributes of each word. Associative memories seem to be a natural way to represent this information. Another possible use occurs in the parsers used with such systems. By including a relatively small number of associative checks in a parsing program, one can attain significant improvements over purely syntatic techniques. This is part of a current trend in both natural and artificial language work to combine syntatic and semantic considerations.

A further use of associative processing would be in helping to extend parsing beyond individual sentences. The improved processing ability may make it feasible to record various associations (like pronoun reference) dynamically in scanning a text. The results of Klein and Simmons [3] using only the attribute "depends on" is very encouraging in this regard. Notice that an application like this would have much higher frequencies of SET and ERASE than we assumed in setting up the simulator.

There are other similar applications in artificial intelligence programming which are fairly obvious and won't be mentioned here. We will, however, consider one possible application to cognitive psychology.

The associative language and its simulator can be thought of as a model for certain human cognitive processes. With a proper choice of experiments one might be able to determine the behavior of humans in tasks similar to program runs. Among the interesting questions are the relative difficulties of F1......F6 and the cost of recursion. Although there have been related experiments, we know of no directly applicable results. It should also be feasible to design an associative simulator whose parameters could be easily changed in an attempt to model experimental results.

Whatever the importance of these applications, they only make use of most obvious features of associative processing. For really imaginative applications we will need a language more powerful than any that presently exists

Besides the obvious extensions mentioned in section I, many improvements could be made in our associative language. One very important feature would be a flexible subroutine system. This will be somewhat difficult to implement for our system because of the key role of bound variables. A related development would be the addition of explicitly recursive statements. For example, statement ③ of Appendix A will only extend the transitivity of ABOVE one step. The difficulty in this case and in general is the same: describing the criterion for stopping recursive processes. Intuitively, the stopping condition is "when nothing new is added" but its complete expression in the general case is not obvious.

A significant increase in expressive power could be attained by adding elaborate declaration facilities to the associative language. One could have a concept of "derived attribute" defined in terms of existing attributes and objects. Such constructs could be used in all the usual ways and several new ones. One could, for example, set an aspiration level determining how much time should be expended in attempting to find an answer. An additional command, INCORPORATE, could be used to write explicitly all associations applying to the derived attribute. If incorporating were allowed to depend on the number of references one could investigate problems of reinforcement and concept formation in the model.

Two less important, if no less perplexing, possibilities are quantifiers and the ordering of lists. We originally felt that logical quantifiers (There Exists, For All) would be essential to the system. In practice no explicit use of any quantifier other than "For All" has occurred. The notion of ordering elements on the multiplicity lists was not included in the original version, but would be useful in certain cases. The point is that whenever an association (e.g., lists order) can be represented directly in the structure this representation is very efficient. However, the associative scheme was designed to make information independent of structure. Rather than order the multiplicity lists one could imbed AL in one of the existing list languages.

The question of imbedding is an important part of any discussion of extensions of AL. Any associative processing techniques, including those described here, are efficient for only a limited set of operations. To make full use of the abilities of associative languages, one will require sophisticated methods for dealing with mixed statements. The simple examples in Appendix A give some indication of how mixed statements arise and how they might be expressed. The imbedding problem is complicated by a plethora of possible host languages. One can envision problems where one or another of these languages is clearly superior to the others. Perhaps the most reasonabl course is to imbed the associative processes in a compiler-compiler such as [1]. Besides allowing various imbeddings, this would facilitate experimentati with the syntax of the associative language.

The present syntax of AL models rather closely the structure of the associative processor. This is a natural way to start, but the source language need not have any close relation to the implementation. The present form is easy to write, but seems difficult to read and, therefore, to debug. There are several other notations, including some from set theory, being considered as alternatives.

Besides the difficulties peculiar to AL, there are several others which apply to any associative processing scheme. Among the foremost is the representation of n+1 – tuples in a memory designed for n – tuples. For examples, we have no direct way to represent

"the number of fingers on a hand is five".

One could break this up into triples such as:

NUMBER FINGERS · (NUMBER) = FINGERS

NUMBER FINGERS · (HAND) = 5

but this solution is obviously quite clumsy. This difficulty is the price one pays for having a uniform, tightly packed structure. Notice that the statement could be handled by a compiler and is primarily an implementation level problem.

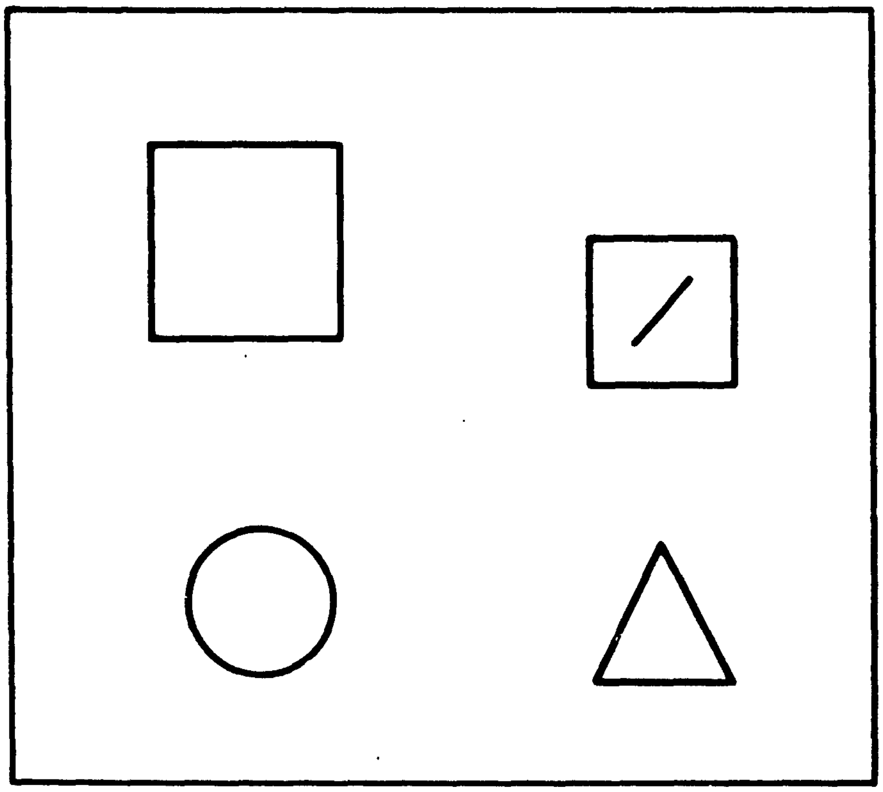
Another problem of implementation concerns the representation of numbers in an associative memory. The alternatives seem to be using an extra bit in each object to mark absolute numbers or using numbers indirectly like IPL-V data terms. Both alternatives are sufficiently unattractive to warrant further work on this problem.

Despite these and other difficulties, we believe that associative processing has an important role to play in current and future research. We have already indicated several reasons for this belief. A further advantage of associative processing is the high degree of parallelism inherent in associative statements. For example, statement ④ of Appendix A applies anywhere in memory where one object is described to be inside another. Although the parallelism is illusory in the simulated scheme, the language itself could be used effectively on a computer which actually did process in parallel.

Most of the previous work on associative processing has been hardwai oriented. A good review of this work through 1963 may be found in Fuller [2]. Some descriptions of more recent work may be found in references [12-15]. In addition, several of the most recently announced computers make use of very small associative memories in their addressing schemes. An interesting statisi cal model of associative memory operation has been described by G. Simmons [9]. The total amount of work on software aspects of associative processing is surprisingly small. It is our hope that this paper will help stimulate researc in this important area of information processing.

3-23-6150

APPENDIX A



PICTURE USED IN SAMPLE RUN

```
SET LEFT•(TRI)=CIRC, LEFT•(TRI)=SQ1
SET LEFT•(SQ2)=CIRC, ABOVE•(CIRC)=SQ2
SET ABOVE•(TRI)=SQ2, ABOVE•(SQ2)=SQ1
SET INSID•(SQ2)=LINE, RUN
①     HOW ARE SQ2 AND CIRC RELATED
TYPE W•(SQ2)=CIRC, W•(CIRC)=SQ2, RUN
             LEFT•(SQ2)=CIRC
             ABOVE•(CIRC)=SQ2

②     WHAT IS ABOVE AND RIGHT OF CIRC
FOR ABOVE•(CIRC)=X TYPE LEFT•(X)=CIRC END, RUN
             LEFT•(SQ2)=CIRC

③     ABOVE IS TRANSITIVE
FOR ABOVE•(X)=Y WHERE ABOVE•(Y)=Z SET ABOVE•(X)=Z END
TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=SQ1

④     INSID HAS INHERITANCE
FOR INSID•(X)=Y WHERE W•(Z)=X SET W•(Z)=Y END
TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=LINE
             ABOVE•(TRI)=SQ1
FOR INSID•(X)=Y WHERE W•(X)=Z IF Y≠Z SET W•(Y)=Z END
TYPE W•(LINE)=Z, RUN
             ABOVE•(LINE)=SQ1
             LEFT•(LINE)=CIRC

SET AREA•(CIRC)=ACIRC, AREA•(SQ1)=ASQ1
SET AREA•(SQ2)=ASQ2, AREA•(TRI)=ATRI
SET AREA•(LINE)=0, RUN
1→ATRI, 2→ASQ1, 2→ACIRC, 3→ASQ2, RUN

⑤     WHAT IS SMALLER THAN SQ1
FOR AREA•(X)=Y IF ASQ1>Y TYPOB X,Y END, RUN
             LINE
             0          000
             TRI
             ATRI       001

⑥     WHAT IS LARGER THAN AND ABOVE TRI
FOR ABOVE•(TRI)=X WHERE AREA•(X)=Y IF Y>ATRI TYPOB X END, RUN
             SQ2
             SQ1

⑦     ERASE THE LINE
ERASE W•(X)=LINE, W•(LINE)=X, TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=SQ1
```

APPENDIX B

BACKUS NORMAL FORM SYNTAX OF THE ASSOCIATIVE LANGUAGE

This appendix contains a Backus Normal Form (BNF) approximation to the formal syntax of the Associative Language (AL) as used in this paper. The word approximation arises from the inability of BNF to express the notion of bound variables which is so crucial to AL. The other weaknesses of BNF grammars [1] apply in the usual way.

The syntax presented here is of use solely in determining what constructs are presently implemented. Not even the obvious extensions mentioned in Sections I and II have been included. One point that we have tried to establish in the text is that the precise form of any additions to AL is not well determined.

APPENDIX B

ASSOCIATIVE LANGUAGE

```
< integer > :: = < digit > | < integer > < digit >

< name > :: = < letter > | < name > < letter > | < name > < digit >

< object name > :: = < name >

< attribute name > :: = < name > .

< object variable > :: = W |X |Y |Z

< attribute variable > :: = W· |X· |Y· |Z·

< object > :: = < object name > | < object variable > | < integer >

< attribute > :: = < attribute name > | < attribute variable >

< form > :: = < attribute > ( < object > ) = < object >

< closed form > is a form with all variables bound

< statement > :: = < type statement > | < erase statement > < |

                   < typob statement > | < set statement > | < conditional > |

                   RUN | < replacement >

< type statement > :: = TYPE < form > | < type statement >, < form >

< erase statement > :: = ERASE < form > | < erase statement >, < form >

< typob statement > :: = TYPOB < object > | < typob statement >, < object >

< set statement > :: = SET < closed form > | < set statement >, < closed form >

< conditional > :: = < head > < tail >

< head > :: = FOR < form > | < head > WHERE < form > | < head > IF < relation >

                   < head > AND < form > | < head > AND < relation > | IF < relat:

< tail > :: = < statement sequence > END

< statement sequence > :: = < statement > | < statement sequence >, < statement

< replacement > :: = < sum > → < object >

< sum > :: = < object > | < sum > + < object > | < sum > - < object >

< relation > :: = < sum > = < object > | < sum > > < object > | < sum > Δ < ob,
```

BIBLIOGRAPHY

1. Feldman, J., "A Formal Semantics for Computer-Oriented Languages", Doctoral Dissertation, Carnegie Institute of Technology, 1964.

2. Fuller, R. H., "Content-Addressable Memory Systems", UCLA, Dept. of Engineering Report, 63-25

3. Klein, S. and R. Simmons, "Syntatic Dependence and the Computer Generation of Coherent Discourse", Mechanical Translation, 1963.

4. Katz, J. and P. Postal, An Integrated Theory of Linguistic Descriptions, Cambridge, MIT Press, 1964.

5. Newell, A., ed., Information Processing Language-V Manual, Prentice Hall, Englewood Cliffs, New Jersey, 1961.

6. Newell, A., "A Note on the Use of Scrambled Addressing for Associative Memories", Unpublished paper, December 1962.

7. Roberts, L. G., "Graphical Communication and Control Languages", Second Congress on Information System Science, Hot Springs, Va., 1964.

8. Robinson, Jane,"Automatic Parsing and Fact Retrieval, Rand Memorandum RM-4005-PR.

9. Simmons, G. J., "A Mathematical Model for an Associative Memory", Sandia Corporation Report, SCR-641, April 1963.

10. Simmons, R. F., "Answering English Questions by Computer", Comm. ACM V8 #1, January 1965.

11. Thompson, F. B., et al, "Deacon Breadboard Summary", General Electric Company, Santa Barbara, RM 64TMP-9.

12. Ewing, R. G. and P. M. Davies, "An Associative Processor", Proc. IFIPS 1964 Fall Joint Computer Conference.

13. Gall, R. G, "A Hardware-Integrated GPC/Search Memory", Proc. IFIPS 1964 Fall Joint Computer Conference.

14. McAteer, J. E., J. A. Capobianco and R. L. Koppel, "Associative Memory System Implementation and Characteristics", Proc. IFIPS 1964 Fall Joint Computer Conference.

15. Raffel, J. I. and T. S. Crowther, "A Proposal for an Associative Memory Using Magnetic Films", IEEE Trans. on Electronic Computers, EC-13, No. 5, 1964.

APPENDIX A

```
SET LEFT•(TRI)=CIRC, LEFT•(TRI)=SQ1
SET LEFT•(SQ2)=CIRC, ABOVE•(CIRC)=SQ2
SET ABOVE•(TRI)=SQ2, ABOVE•(SQ2)=SQ1
SET INSID•(SQ2)=LINE, RUN
①    HOW ARE SQ2 AND CIRC RELATED
TYPE W•(SQ2)=CIRC, W•(CIRC)=SQ2, RUN
             LEFT•(SQ2)=CIRC
             ABOVE•(CIRC)=SQ2

②    WHAT IS ABOVE AND RIGHT OF CIRC
FOR ABOVE•(CIRC)=X TYPE LEFT•(X)=CIRC END, RUN
             LEFT•(SQ2)=CIRC

③    ABOVE IS TRANSITIVE
FOR ABOVE•(X)=Y WHERE ABOVE•(Y)=Z SET ABOVE•(X)=Z END
TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=SQ1

④    INSID HAS INHERITANCE
FOR INSID•(X)=Y WHERE W•(Z)=X SET W•(Z)=Y END
TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=LINE
             ABOVE•(TRI)=SQ1
FOR INSID•(X)=Y WHERE W•(X)=Z IF YΔZ SET W•(Y)=Z END
TYPE W•(LINE)=Z, RUN
             ABOVE•(LINE)=SQ1
             LEFT•(LINE)=CIRC

SET AREA•(CIRC)=ACIRC, AREA•(SQ1)=ASQ1
SET AREA•(SQ2)=ASQ2, AREA•(TRI)=ATRI
SET AREA•(LINE)=0, RUN
1→ATRI, 2→ASQ1, 2→ACIRC, 3→ASQ2, RUN

⑤    WHAT IS SMALLER THAN SQ1
FOR AREA•(X)=Y IF ASQ1>Y TYPOB X,Y END, RUN
             LINE
             0         000
             TRI
             ATRI      001

⑥    WHAT IS LARGER THAN AND ABOVE TRI
FOR ABOVE•(TRI)=X WHERE AREA•(X)=Y IF Y>ATRI TYPOB X END, RUN
             SQ2
             SQ1

⑦    ERASE THE LINE
ERASE W•(X)=LINE, W•(LINE)=X, TYPE ABOVE•(TRI)=X, RUN
             ABOVE•(TRI)=SQ2
             ABOVE•(TRI)=SQ1
```